Tools and Techniques for Softmodem IHV/ISV Self-Validation of Compliance with PC 99 Guidelines for Driver-Based Modems

Revision 1.0
7 July 1998
Intel® Architecture Labs
Intel Corporation

1. Introduction

The PC 99 System Design Guide modem chapter contains a number of recommended guidelines intended to help designers in the development of WDM-based software modem implementations for the Windows* 98 and Windows NT* operating systems. These guidelines were developed because software modems are among the first computationally intensive services where third party vendors provide kernel mode drivers that can have a significant impact on OS scheduling services. As other services move into kernel mode it is envisioned that similar guidelines may need to be established for a range of kernel mode drivers, potentially including low-latency audio, 3D positional audio, music synthesis, software MPEG, and AC-3 DVD decoders.

This document details techniques whereby modem Independent Hardware Vendors (IHVs) and Independent Software Vendors (ISVs) can verify that their implementations conform to the PC 99 guidelines. Some of the instrumentation techniques described in this document will only be realizable by designers with source code availability. There is no intent to document methods for external "black box" testing of modem performance and conformance to the PC 99 guidelines[1]. Any kernel mode service or driver can use the techniques described here and it is hoped that they will be of use to a range of kernel mode services and drivers in the future.

[1] Many of the techniques described in this document rely heavily on processor features specific to the Intel Architecture Pentium® and P6 processor families. The latter includes the Pentium Pro, Pentium II and Celeron™ processors. It may not be feasible to use these techniques on other Intel Architecture processors.

1.1 Pentium II Processor Model Specific Registers

Intel Architecture processors belonging to the Pentium® and P6 processor families define a number of model specific registers[2]. The procedures outlined in this report are correct for P6 family processors, including the Pentium Pro, Pentium II and Celeron™ processors, as of the date of this writing.

Software that executes at privilege level 0 or in real-address mode can read from and write to these model specific registers using the RDMSR and WRMSR instructions, respectively. Certain model specific registers are distinguished in the sense that software, irrespective of the privilege level at which it executes, can read from them using special instructions.

1.1.1 Time Stamp Counter

Intel Architecture processors belonging to the Pentium and P6 processor families provide a model specific register which is incremented every processor clock cycle. This is the time stamp counter, and software which is executing at any privilege level can read it using the RDTSC instruction[3]. The Intel application note, "Using the RDTSC Instruction for Performance Monitoring", explains how to use the time stamp counter to get accurate cycle measurements for small sections of code and average timings for functions. Techniques in this report that use the time stamp counter can all be adapted to the Pentium processor.

In the P6 processor family the time stamp counter is incremented by one (1) every processor clock cycle even when the processor is halted by the HLT instruction or the external STPCLK# pin. Thus it effectively counts clocks during which the processor is *not* in one of the sleep states (i.e., Sleep State and Deep Sleep State) as defined in section 7.2 of the Pentium II Processor Developer's Manual. In the absence of power management (or if power management is restricted to ACPI states G0 and G1 as defined in section 2.2 of the Advanced Configuration and Power Interface Specification), the time stamp counter can be used to measure the elapsed time between two software events with high accuracy. Aggressive power management can be effectively disabled via an idle priority user mode thread that spins indefinitely in a compute-bound loop; C language source code for such an application is provided in Appendix A.

1.1.2 Performance-Monitoring Counters and Events

Intel Architecture processors belonging to the Pentium and P6 processor families provide a small number of model specific performance-monitoring *counters* (2 for the Pentium Pro, Pentium II and Celeron processors). These performance counters can be dynamically configured to count predefined performance-monitoring *events* using the WRMSR instruction to write to either the EVNTSEL0 or the EVNTSEL1 model specific register. On P6 family processors, software that is executing at any privilege level can read either of the performance-monitoring counters using the RDPMC instruction[4].

Intel Architecture processors belonging to the Pentium and P6 processor families define a number of model specific performance-monitoring events. The set of events defined is *not* the same for the two processor families, nor is the set of events the same for all processors within the P6 family[5]. A complete

---

[2] Model specific registers are not guaranteed to be duplicated or provided in subsequent generations of Intel Architecture processors, and may be changed or removed without notice.

[3] If the time stamp disable flag (TSD) in the CR4 register is set then software must be executing at privilege level 0 to use the RDTSC instruction. Windows operating systems typically do not set this flag.

[4] If the performance counter enable (PCE) flag in the CR4 register is not set then software must be executing at privilege level 0 to use the RDPMC instruction.

[5] Events defined for current processor families are not guaranteed to be duplicated or provided in future generations of Intel Architecture processors.

list of events for both processor families, including new events for the Pentium II processor, can be found in Volume 3 of the Intel Architecture Software Developer's Manual.

The following Pentium II processor performance-monitoring events are of particular interest for the techniques described in this document:

- 79H (CPU_CLK_UNHALTED): Number of cycles during which the processor is *not* halted.
- C0H (INST_RETIRED): Number of instructions retired.
- C6H (CYCLES_INT_MASKED): Number of cycles during which interrupts are masked.

When the CYCLES_INT_MASKED event is selected by a performance counter, the performance counter counts the cumulative number of processor cycles during which interrupts are masked (i.e., disabled). This event is defined for all processors in the P6 processor family but is not defined for the Pentium processor. To obtain an estimate of the number of instructions executed while interrupts are masked the other performance counter can be set to count the INST_RETIRED event. To do this read both counters and the time stamp register at two points in time, for example, at two ticks of the Programmable Interval Timer (PIT). The number of instructions executed while interrupts were disabled during the interval can then be estimated by multiplying the difference in the INST_RETIRED counts by the difference in the CYCLES_INT_MASKED counts and dividing by the difference in the time stamps[6].

2. Suggested Techniques by Guideline

This section presents an overview of applicable techniques organized to match the Driver Based Modem Guidelines in the PC 99 System Design Guide modem chapter. Detailed explanations of how to implement each technique, including pseudocode in many cases, are given in section 3.

2.1 Driver-based modem processor usage is not excessive (19.27)

19.27.1 Driver minimum-maximum cycle times are appropriate. Conformance to this guideline is best verified by design review. PC 99 recommends that driver based modems use cycle times in the range 3 to 16 milliseconds. If a driver-based modem is not designed to conform to this guideline it may not be realistic to bring it into conformance during implementation.

19.27.2 Average processor usage during data transmission does not exceed 25 percent. Conformance to this guideline is best verified using a profiling tool such as Intel's VTune performance analysis and tuning environment. Because usage includes system calls made by the driver, as well as operating system overhead incurred to schedule deferred procedure calls (DPCs) and threads, profiling measurements should be made using differencing techniques that compare driver plus operating system usage in active and inactive states. Section 3.1 gives detailed instructions for performing difference-based profiling using VTune.

19.27.3 Total processor usage during data transmission does not exceed 50 percent. Conformance to this guideline can be verified by instrumenting the driver to record the time of entry and exit to all driver-based modem operating system schedulable entities. An approach using the highly accurate time stamp counter on Intel Architecture processors is described in detail in section 3.2.

19.27.4 Average processor usage in retrain mode does not exceed 50 percent. Conformance to this guideline is best verified using a profiling tool such as VTune. Because usage includes system calls made by the driver, as well as operating system overhead incurred to schedule deferred procedure calls (DPCs) and threads, profiling measurements should be made using differencing techniques that compare driver

[6] This approach implicitly estimates the clocks per instruction (CPI) over the interval by dividing the difference in the time stamps by the difference in the instructions retired counts.

plus operating system usage in active and inactive states. Section 3.1 gives detailed instructions for performing difference-based profiling using VTune.

19.27.5 Total processor usage in retrain mode does not exceed 75 percent. Conformance to this guideline can be verified by instrumenting the driver to record the time of entry and exit to all driver-based modem operating system schedulable entities. An approach using the highly accurate time stamp counter on Intel Architecture processors is described in detail in section 3.2.

2.2 Driver does not disable interrupts for excessive periods of time (19.28)

On Intel Architecture processors implementers can verify that their driver meets this guideline by constructing an interrupts masked detection (IMD) driver which configures one of the processor's performance monitoring counters to count cycles during which interrupts are masked (for P6 family processors, C6H, CYCLES_INT_MASKED). Such a driver could either poll the counter or load it with a value (using WRMSR instruction) and catch the non-maskable interrupt that will be generated when the counter overflows. Pseudocode for one possible driver design is given in section 3.3.

Interpretation of the output from such an IMD driver requires careful analysis. If the IMD driver detects episodes where interrupts are disabled for excessive periods of time when the modem is in use, the IMD driver should be used to check whether such episodes also occur on the same hardware configuration when the driver is not in use. Detection of an equal number of episodes of similar size when the modem is not in use would, in general, tend to rule out the modem driver. However, lack of detection of such episodes when the modem is not in use would not necessarily implicate the modem because a combination of drivers could together disable interrupts for an excessive period of time.

On Intel Architecture processors such episodes can be disambiguated by a driver which catches the non-maskable interrupt generated by a performance-monitoring counter that has been configured to count cycles during which the processor is *not* halted (for P6 family processors, 79H, CPU_CLK_UNHALTED). The driver's interrupt handler stores the instruction pointer in a circular buffer for retrieval after the fact by the IMD driver and reinitializes the performance-monitoring counter to a short interval (e.g., 10 microseconds). Pseudocode for one possible driver design is given in section 3.4. Keep in mind, however, that a form of the famous Heisenberg uncertainty principle applies here. Because of the frequency at which non-maskable interrupts are generated, this second technique imposes significant overhead and will likely have a noticeable impact on system performance in general and the performance of the softmodem in particular. Thus, what is being measured is being measured quite accurately, but unfortunately it is not what one ultimately wishes to measure, which is the behavior of the system when measurements are not being made. This caveat does *not* apply to the IMD driver technique, but only to the disambiguating technique described in this paragraph.

2.3 Driver handles thread priorities appropriately (19.29)

19.29.1 Driver uses thread priorities 28 and above. Conformance to this guideline is best verified by design review. PC 99 recommends that driver based modems use priorities range 28 through 30. Thread priority 31 should be reserved for short-duration time-critical processing by the operating system. Non-modem thread-based drivers should use thread priorities 27 and lower. If a driver-based modem is not designed to conform to this guideline it may not be realistic to bring it into conformance during implementation.

19.29.2 Driver limits execution of simultaneously enqueued DPCs. Conformance to this guideline is best verified by design and if a driver-based modem is not designed to conform to this guideline then it may not be realistic to bring it into conformance during implementation. A polling driver that tests to determine the execution time required by all the DPCs in the queue can be implemented by creating two 1 millisecond periodic timers, the first of which has an associated HighImportance CustomDpc, and the

second of which has an associated MediumImportance (the default) CustomDpc. By reading the processor time stamp register in each DPC and differencing the reading, the driver can determine the length of the DPC queue when the PIT timer fired. Pseudocode for one possible design is given in section 3.5.

19.29.3 Driver does not disable thread preemption for excessive periods of time. On Intel Architecture processors implementers can instrument their modem driver code to use the processor's time stamp counter to record the start and finish time of each DPC. A high priority driver thread can then be used to detect instances where thread preemption was disabled by waiting on a 3 millisecond periodic timer and checking the total elapsed time in DPCs since the last time the thread woke up. Pseudocode for one possible design is given in section 3.6.

2.4 Driver tolerates reasonable operating system and bus latencies (19.30)

19.30.1 Driver tolerates interrupt latency. A driver-based modem should be able to tolerate a period with interrupts disabled of 2 milliseconds. This guideline is best verified by testing. On Intel Architecture processors under Windows 98 a test driver can be implemented by spinning for 2 milliseconds while reading the processor's time stamp register in a tight loop with interrupts disabled. On Intel Architecture processors under Windows NT a test driver can execute the same loop at DIRQL for the modem device or at CLOCK2_LEVEL if the softmodem uses a system timer. C language source code for a function that implements the loop is given in appendix C.

19.30.2 Driver tolerates DPC latency. A driver-based modem should be able to tolerate a continuous period of 5 milliseconds during which a DPC which it has enqueued is held-off from execution (possibly by other DPCs). This guideline is best verified by testing. On Intel Architecture processors a test driver is easily implemented by creating a CustomDpc routine which spins for 5 milliseconds while reading the processor's time stamp register in a tight loop. The CustomDpc is associated with a timer, which can be periodic. The CustomDpc must reset a non-periodic timer after spinning, but such timers do offer the advantage that there can be a random delay before the timer expires. C language source code for a function that implements the loop is given in appendix C.

19.30.3 Driver tolerates thread latency. A WDM driver-based modem should be able to tolerate a 12-millisecond period during which thread scheduling is continuously disabled. This guideline is best verified by testing. On Intel Architecture processors a test driver is easily implemented by creating a driver routine which spins for 12 milliseconds while reading the processor's time stamp register in a tight loop. A driver created kernel mode real-time thread executing at THREAD_PRIORITY_TIME_CRITICAL calls this routine, optionally at raised IRQL (e.g., DISPATCH_LEVEL). C language source code for the driver routine is given in appendix C.

19.30.4 Driver tolerates PCI bus latency. A WDM driver-based modem should be able to tolerate a 100-microsecond hold-off from access to the PCI bus caused by other bus masters. This guideline is best verified by testing with a programmable PCI device. If a programmable PCI device of this type is not available, a possible approach to testing would be to install a PCI bus mastering device and load a driver that is known to not be compliant with the PCI standard revision 2.1.

3. Details for Selected Techniques

3.1 Profiling Using VTune

Intel's VTune tool is an ideal tool for obtaining highly accurate profiling measurements of utilization on a per module basis. Because total usage includes system calls made by the driver as well as operating system overhead incurred to service hardware interrupts and schedule system services such as deferred procedure calls (DPCs) and threads, profiling measurements should ideally be made using differencing

techniques in order to capture total utilization. The Idle Priority User Mode MIPS Eating Application (IdleApp) from appendix A can be used to soak up idle processor cycles, preventing them from being erroneously attributed to the kernel and simplifying the calculation of total utilization.

VTune should be configured so as to minimize its load on the system while measurements are being made by closing any unneeded windows such as the VTune Assistant. Sampling is best made using the Real-Time Clock (RTC) but can also be made using Time Based Sampling (TBS). The following configuration options should be selected from the Project Options menu:

- Sampling Tab: "Sampling Interval" should be 0.1 ms or any *prime* multiple of 0.1 (e.g., 0.2, 0.3, 0.5, 0.7, etc.) to minimize effects of aliasing; "How long to run" should be sufficient to generate 100K samples at a minimum.

- Advanced Tab: RTC or TBS sampling; "Sample BufferSize" should be 1024 KB minimum in order to minimize the effects of file I/O on the system.

- Miscellaneous Tab: "Use Meter during Sessions to Indicate Progress" should *not* be selected; "Automatically increment the Session ID" should be selected.

The procedure to measure the utilization of the modem driver is as follows:

1. Open and configure VTune as specified above.
2. Launch the Idle App (see appendix A).
3. Run VTune.
4. Start the driver.
5. Run VTune again.
6. Stop the driver.
7. Repeat steps 3 through 6 as needed to achieve repeatable results.

The total driver utilization is given by the difference between the measured utilization of the Idle App with and without the driver running provided that the Idle App runs frequently enough that it is not paged out. This should not be a problem if Idle App usage is over 10%.

3.2 Worst Case Driver Execution Times Using the Time Stamp Counter

Implementers can measure the worst case execution time inclusive of system calls made by the driver but exclusive of operating system overhead incurred to schedule system services, by instrumenting the driver to record the time of entry and exit to all driver-based modem functions. It is also necessary to record the times at which driver threads wait for synchronization objects (an exit) and have their waits satisfied (an entry). On Intel Architecture processors, highly accurate timings can be obtained at minimal cost using the processor's timestamp counter, minimizing the overhead of collecting measurements.

To implement this technique a macro can be defined that resolves to either: 1) a read of the time stamp register and write to a buffer or 2) whitespace (as is commonly done with debugging output). A low priority thread can either write out the buffer for post analysis or periodically sweep through the buffer looking for execution times over 50% over the relevant cycle time. Nested pairs of entry and exit times can be discarded, leaving the entry and exit times to those driver entities that were scheduled by the kernel.

3.3 Interrupts Masked Detection Driver

The pseudocode below outlines a driver that detects possible occurrences of interrupts being masked for periods in excess of 1 millisecond. This driver can give false positive output but is relatively simple to construct. If the total number of cycles during which interrupts are disabled is low, this driver will suffice to verify compliance with this guideline. C language code for the DpcRoutine for this driver is given in appendix B. Because this driver relies on the `CYCLES_INT_MASKED` performance-monitoring event this driver cannot be implemented on the Pentium processor.

```
• In the DriverEntry routine do the following:
    Set the interrupt interval for the programmable interval timer (PIT)
       to 1 millisecond.
    Initialize a timer and a custom DPC routine (DPC pseudocode below).

• In the Create dispatch routine do the following:
    Configure a performance counter to count CYCLES_INT_MASKED (C6H)
       using WRMSR to write to the EVNTSEL0 or EVNTSEL1 register.

• In the Close dispatch routine do the following:
    Stop the performance counter by clearing the EVNTSEL register.

• In the Read dispatch routine do the following:
    Set the timer to be recurring with a period of 1 millisecond and
       with an associated custom DPC routine.
    Set the IRP to be cancelable, mark it pending and save a pointer to
       it in the DEVICE_EXTENSION.
    Return STATUS_PENDING.

• In the custom DPC routine do the following:
    Read the timestamp and then the cycles with interrupts disabled.
    Calculate the deltas in both since the DPC last ran.
    If the sum of the delta in cycles with interrupts disabled plus the
       saved delta in cycles with interrupts disabled exceeds 100
       microseconds
    Then report a possible occurrence (i.e., complete the IRP) and
       cancel the timer.
    Else update the saved time stamp and cycle count and delta with the
       current (new) values and return without touching the IRP.
```

As stated previously, the above driver can give spurious false positive readings. A more accurate driver would load the performance counter with a value (using WRMSR instruction) and catch the non-maskable interrupt that will be generated when the counter overflows. By loading the counter with a value equal to –50 microseconds in clocks (e.g., -15000 at 300 MHz) the driver could accurately detect periods of 100 microseconds with interrupts disabled by checking the interval between successive counter overflows. Any instance of interrupts disabled for at least 100 microseconds will cause 1 pair of 2 successive overflows to be separated by exactly 50 microseconds. Although less likely to generate false positives, this driver can also do so. These can be minimized by reducing the magnitude of the value loaded into the driver, thereby increasing the number of consecutive overflows that a period of 100 microseconds with interrupts disabled must generate. For example, with a value equal to –25 microseconds in clocks, 3 successive pairs (i.e., 4 overflows in all) must each be separated by exactly 25 microseconds.

3.4 CS:EIP Version of Interrupts Masked Detection Driver

The pseudocode below outlines how to modify the Interrupts Masked Detection driver to detect which driver is disabling interrupts. The modified driver catches the non-maskable interrupt generated by a performance-monitoring counter that has been configured to count cycles during which the processor is *not* halted (for the P6 processor family, 79H, CPU_CLK_UNHALTED). The driver's interrupt handler stores the instruction pointer in a circular buffer for retrieval after the fact by the IMD driver and

reinitializes the performance-monitoring counter to a short interval. Keep in mind, however, that with such a high interrupt rate the effects of interrupt servicing on system performance in general and the performance of the softmodem in particular will be non-negligible. Again, because the Interrupts Masked Detection driver relies on the CYCLES_INT_MASKED performance-monitoring event, this driver cannot be implemented on the Pentium processor.

- To the DriverEntry routine add the following:
  Allocate a circular buffer for the NMI handler in locked memory and a pointer to the buffer. Each slot in the buffer will receive the contents of the CS, EIP and Time Stamp Counter registers.

- To the Create dispatch routine add the following:
  Configure a second performance counter to count CPU_CLK_UNHALTED (79H) using WRMSR to write to the other EVNTSEL register
  After configuring the performance counter install an NMI Handler.
  Write zero (0) minus the desired NMI cycle time to the performance counter (e.g., -3000 for 10 microseconds at 300 MHz).

- To the Close dispatch routine add the following:
  Stop the second performance counter by clearing the corresponding EVNTSEL register.
  Remove the NMI Handler.

- In the Unload routine do the following:
  Release the circular buffer and pointer.

- In the NMI handler do the following:
  Read the Time Stamp Counter.
  Increment the pointer to the next slot in the circular buffer.
  Read CS and EIP from the stack and write them and the Time Stamp Counter value to the pointed to slot in the circular buffer.
  Write zero (0) minus the desired NMI cycle time to the performance counter (e.g., -3000 for 10 microseconds at 300 MHz).

- To the custom DPC routine add the following:
  If the sum of the delta in cycles with interrupts disabled plus the saved delta in cycles with interrupts disabled exceeds 100 microseconds
  Then report a possible occurrence (i.e., complete the IRP). Return the last 1000 microseconds of entries from the circular buffer together with the sum of the delta in cycles.

3.5 DPC Queue Length Polling Driver

The pseudocode below outlines a polling driver that periodically tests to determine the execution time required by all the DPCs in the queue. Because this driver polls to determine the DPC queue length it is possible for it to miss an instant in time where the execution time required by all DPCs in the queue is over 500 microseconds. The driver relies on periodic timers and a HighImportance CustomDpc. It is therefore only portable to Windows NT 4.0 and higher as well as Windows 98.

- In the DriverEntry routine do the following:
  Set the interrupt interval for the programmable interval timer (PIT) to 1 millisecond.
  Initialize a timer and a MediumImportance custom DPC routine.
  Initialize another timer and a HighImportance custom DPC routine.

- In the HighImportance custom DPC routine do the following:
  Read the Time Stamp Counter.
  Check the MediumImportanceDpcRan flag in the driver extension to avoid a possible race condition between the timers.
  If the MediumImportance DPC did *not* already run

```
   Then save the Time Stamp Counter value in the device extension and
      set the HighImportanceDpcRan flag in the device extension.
   Else (i.e., if it did run) clear the MediumImportanceDpcRan flag in
      the device extension.
```

- In the MediumImportance custom DPC routine do the following:

```
   Read the Time Stamp Counter.
   Check the HighImportanceDpcRan flag in the driver extension to avoid
      a possible race condition between the timers.
   If the HighImportance DPC already ran
   Then subtract the saved time stamp (from device extension) from the
      current time stamp and clear the HighImportanceDpcRan flag in the
      device extension.
      If the difference is over 500 microseconds in clocks
      Then report an occurrence (i.e., complete the IRP) and cancel
         both timers.
      Else do nothing.
   Else (i.e., if it did not run) set the MediumImportanceDpcRan flag
      in the device extension.
```

- In the Create dispatch routine do the following:

```
   Set both timers to be recurring with a period of 1 millisecond.
```

- In the Close dispatch routine do the following:

```
   Cancel both timers.
```

- In the read dispatch routine do the following:

```
   Set the IRP to be cancelable, mark it pending and save a pointer to
      it in the DEVICE_EXTENSION.
   Return STATUS_PENDING.
```

3.6 Thread Scheduling Disabled Driver

The pseudocode below outlines a driver that detects possible occurrences of thread scheduling being disabled by an instrumented modem driver for periods in excess of 3.3 milliseconds. This driver assumes that the modem driver itself has been instrumented to use the processor's time stamp counter to record the start and finish time of each DPC and that the time stamps are in a circular buffer in page locked memory whose address is made available to this driver.

- In the DriverEntry routine do the following:

```
   Set the interrupt interval for the programmable interval timer (PIT)
      to 1 millisecond.
   Initialize a timer.
   Create a system thread.
```

- In the Thread's routine do the following:

```
   Set priority to 31.
   Wait on the timer.
   Read the Time Stamp Counter and then calculate the cycles in DPCs
      since the last execution of this step.
   Check the calculated DPC execution time (in cycles) to see if thread
      scheduling has been disabled excessively by DPCs.
```

- In the Create dispatch routine do the following:

```
   Obtain a handle to the region of page locked memory which is used by
      the modem driver to record when its DPC begins and completes
      execution.
```

- In the Read dispatch routine do the following:

```
   Set the timer to be recurring with a period of 3 milliseconds.  Set
      the IRP to be cancelable, mark it pending and save a pointer to
      it in the DEVICE_EXTENSION.
```

```
Return STATUS_PENDING.
```

4. Conclusion

By using the Pentium Pro, Pentium II and Celeron processor time stamp counter and performance counters, self-validation of conformance to the PC 99 driver-based modem guidelines is eminently feasible. We have presented detailed tools and techniques which softmodem IHVs/ISVs can use to validate that their modem meets each of the PC 99 softmodem execution guidelines. A few techniques are documented as detailed instructions on the use of VTune. Most techniques are documented with detailed pseudocode, some for free standing drivers, others for drivers which function in conjunction with instrumentation in the softmodem driver itself. For selected techniques detailed source code is provided in the appendices. All but one technique, the Interrupts Masked Detection driver, can also be implemented on the Pentium processor.

5. Bibliography

Advanced Configuration and Power Interface Specification, rev 1.0. Intel Corporation, Microsoft Corporation and Toshiba Corporation, 1996. URL: http://www.teleport.com/~acpi/spec.htm

Intel Architecture Software Developer's Manual, Volume 2: Instruction Set Reference Manual. Intel Corporation, 1998. URL: http://www.intel.com/design/pentiumii/manuals/243191.htm

Intel Architecture Software Developer's Manual, Volume 3: System Programming Guide. Intel Corporation, 1998. URL: http://www.intel.com/design/pentiumii/manuals/243192.htm

Survey of Pentium Processor Performance Monitoring Capabilities & Tools. Intel Corporation, 1998. URL: http://developer.intel.com/drg/mmx/AppNotes/PERFMON.HTM

Using the RDTSC Instruction for Performance Monitoring. Intel Corporation, 1998. URL: http://developer.intel.com/drg/pentiumII/appnotes/RDTSCPM1.HTM

VTune (Visual Tuning Environment), Intel Corporation. URL: http://developer.intel.com/design/perftool/vtune/index.htm

Appendix A: An Idle Priority User Mode MIPS Eating Application

```
/*M*
//                  INTEL CORPORATION PROPRIETARY INFORMATION
//             This software is provided by Intel Corporation for your
//          internal use only for the purposes of performance measurement
//               and may not be otherwise used, copied or disclosed.
//       Copyright (c) 1997, 1998 Intel Corporation. All Rights Reserved.
//
//          The receipt of or possession of this program does not convey
//            any rights to reproduce its contents or to manufacture,
//            use, or sell anything that it may describe, in whole, or
//            in part, without specific consent of Intel Corporation.
//
//  Module name:
//      IdleAppMain.c
//
//  PVCS:
//      $Workfile$
//      $Revision$
//      $Modtime$
//
//  Revision History:
//      03-11-98  ECR  Cleaned up, formatted appropriate for external release
//      09-12-97  ECR  Spins at Idle priority (previously spun at 6 because
//                     compute bound)
//      03-11-97  ECR  Initial version
//
//  Purpose: A Win32* console application which spins at idle priority until
//           terminated.  Intended to be used in conjunction with VTune to
//           determine total CPU usage of another application across all
//           modules by differencing VTune usage of this app with/without
//           other app running.
//
//  * Third-party brands and names are the property of their respective
owners.
//
//  Contents: main                   App main.
//
//  Compiler Considerations: Compile without optimization.
//
*M*/

#include <stdio.h>
#include <windows.h>

/*F*
//  Name:       main
//  Purpose:    App Main.  Set priority to idle, then loop soaking up cycles.
//  Context:    It's main().
//  Returns:    1
//  Parameters:
//      IN argc     # of command line args
//      IN argv     command line args
*F*/

int main(
    int cArgc,
    char *pcaArgv[])
{
    DWORD dwI=0;
    DWORD dwJ=0;

    if (SetPriorityClass(GetCurrentProcess(),IDLE_PRIORITY_CLASS) != TRUE) {
        printf("Warning: can't set priority class to idle\n");
    }
```

```
    if (SetThreadPriority(GetCurrentThread(),THREAD_PRIORITY_IDLE) != TRUE) {
        printf("Error: can't set thread priority to idle\n");
        exit(1);
    }else {
        printf("Spinning at idle priority. ^C to exit...");
    } /* if */

    while (1) {
        for (dwI=0; dwI<10000; ++dwI) {
            dwJ = dwI-1;
        } /* for */
        for (; dwJ>0; --dwJ) {
            dwI = dwJ+1;
        } /* for */
    } /* while */

    printf("\n...exiting.");

    return (1);
}
```

Appendix B: Interrupts Masked Detection Driver DPC routine

```
/*M*
//                    INTEL CORPORATION PROPRIETARY INFORMATION
//             This software is provided by Intel Corporation for your
//          internal use only for the purposes of performance measurement
//               and may not be otherwise used, copied or disclosed.
//       Copyright (c) 1998 Intel Corporation. All Rights Reserved.
//
//          The receipt of or possession of this program does not convey
//            any rights to reproduce its contents or to manufacture,
//            use, or sell anything that it may describe, in whole, or
//            in part, without specific consent of Intel Corporation.
//
//  Module name:
//      IMDDriver.c
//
//  PVCS:
//      $Workfile$
//      $Revision$
//      $Modtime$
//
//  Revision History:
//      06-15-98  ECR Initial version
//
//  Purpose: A kernel mode driver which performs a quick and dirty check for
//           possible occurrences of interrupts being disabled for > 100 us.
//           continuously or > 200 us. In 1 ms.  Check consists of summing
//           cycles with interrupts disabled during adjacent periods of
//           ~ 1 millisecond.  If no sum exceeds 100 us. then no occurrences.
//           Converse is NOT true, this test will produce many false positives
//
//  Contents: IMDDpcRoutine
//
//  Compiler Considerations: Only for Pentium(R) Pro, Pentium II, Celeron(TM)
//                           processors.
*M*/

/* . . . */

typedef struct _DEVICE_EXTENSION {
      PDEVICE_OBJECT  pDeviceObject;
      KTIMER          timer;
      PIRP            psavedIrp;
      KDPC            dpcObject;
      LARGE_INTEGER   lastCyclesIntsDisabled;
      LARGE_INTEGER   lastCyclesIntsDisabledDelta;
      LARGE_INTEGER   lastTimeStamp;
      LARGE_INTEGER   OneMillisecondInClocks;
      LARGE_INTEGER   OneHundredMicroseconds;
      LARGE_INTEGER   OneHundredMicrosecondsInClocks;
}DEVICE_EXTENSION, *PDEVICE_EXTENSION;

/* . . . */

/*F*
//  Name:       IMDD_DpcRoutine
//  Purpose:    Handle DPC from Timer Interrupt
//  Context:    Queued by Timer ISR.
//              DriverEntry sets PIT interrupt interval to 1 ms., initializes
//              a periodic timer with 1 millisecond period and this DPC.
//
//              OneHundredMicrosecondsInClocks, OneMillisecondInClocks are
//              set by DriverEntry
//
//  Returns:    void
```

```
//  Design:     Driver completes IRP only if a possible excessive interrupts
//              disabled event is detected.  App provides a single IRP (i.e.,
//              it does a read) and waits indefinitely (no news is good news)
//  Parameters:
//      IN PKDPC Dpc                DPC
//      IN PVOID DeferredContext    DPC context
//      IN PVOID SystemArgument1    ...
//      IN PVOID SystemArgument2    ...
*F*/
VOID
IMDDpcRoutine(
    IN PKDPC Dpc,
    IN PVOID DeferredContext,
    IN PVOID SystemArgument1,
    IN PVOID SystemArgument2)
{
    PDEVICE_EXTENSION pde = (PDEVICE_EXTENSION) DeferredContext;
    PIRP Irp = pde->psavedIrp;
    LARGE_INTEGER CyclesIntsDisabled;
    LARGE_INTEGER CyclesIntsDisabledDelta;
    LARGE_INTEGER LimitInClocks;
    LARGE_INTEGER TimeStamp;
    LARGE_INTEGER TimeStampDelta;
    ULONG Lo;
    LONG Hi;

/* Read the timestamp and then the cycles with interrupts disabled */

    _asm {
        push eax
        push edx
        _emit 0x0f
        _emit 0x31
        mov Lo, eax
        mov Hi, edx
        pop edx
        pop eax
    } /* _asm */

    TimeStamp.LowPart = Lo;
    TimeStamp.HighPart = Hi;

    _asm {
        push eax
        push edx
        push ecx
        xor ecx, ecx
        _emit 0x0f
        _emit 0x33
        mov Lo, eax
        mov Hi, edx
        pop ecx
        pop edx
        pop eax
    } /* _asm */

    CyclesIntsDisabled.LowPart = Lo;
    CyclesIntsDisabled.HighPart = Hi;

    TimeStampDelta.QuadPart = TimeStamp.QuadPart - pde-
>lastTimeStamp.QuadPart;
    CyclesIntsDisabledDelta.QuadPart = CyclesIntsDisabled.QuadPart –
                                       pde->lastCyclesIntsDisabled.QuadPart;

/* If, due to jitter, it's been less than 1 millisecond since we last awoke */
/* then we must assume that all cycles after this for rest of 1 millisecond */
```

```
/* have interrupts disabled, so we adjust LimitInClocks to be that much less
*/
/* than 100 us. If this makes LimitInClocks < 0 then skip to next millisecond
*/

    if (TimeStampDelta.QuadPart < pde->OneMillisecondInClocks.QuadPart) {
        LimitInClocks.QuadPart = pde->OneHundredMicrosecondsInClocks.QuadPart
-
            (pde->OneMillisecondInClocks.QuadPart - TimeStampDelta.QuadPart);
    }else {
        LimitInClocks = pde->OneHundredMicroseconds;
    } /* if */

    if (LimitInClocks.QuadPart < 0) {
        pde->lastCyclesIntsDisabledDelta.QuadPart +=
            CyclesIntsDisabledDelta.QuadPart;
        return;
    }else {
        pde->lastTimeStamp = TimeStamp;
        pde->lastCyclesIntsDisabled = CyclesIntsDisabled;
    } /* if */

    if (CyclesIntsDisabledDelta.QuadPart +
        pde->lastCyclesIntsDisabledDelta.QuadPart >
            LimitInClocks.QuadPart) {
        IoAcquireCancelSpinLock(& Irp->CancelIrql);

        /* We've detected a possible event, so return IRP with data */

        if (Irp->Cancel == TRUE) {
            IoReleaseCancelSpinLock(Irp->CancelIrql);
            /* Driver has been closed, so just return */
            return;
        } /* if */

        Irp->CancelRoutine = NULL;
        IoReleaseCancelSpinLock(Irp->CancelIrql);

        Irp->AssociatedIrp.SystemBuffer =
            (PVOID)((LONGLONG)(CyclesIntsDisabledDelta.QuadPart +
                               pde->lastCyclesIntsDisabledDelta.QuadPart));

        Irp->IoStatus.Status = STATUS_SUCCESS;
        Irp->IoStatus.Information = sizeof(LARGE_INTEGER);
        IoCompleteRequest (Irp, IO_NO_INCREMENT);
    }else {
        pde->lastCyclesIntsDisabledDelta = CyclesIntsDisabledDelta;
    } /* if */
} /* IMDDpcRoutine */
```

Appendix C: Function that Spins Reading the Time Stamp Counter

Below is a C language source code for a function that implements a loop which spins until a given time is reached. This function can be called with interrupts disabled, in a DPC or in a priority 31 thread to test the robustness of the modem driver to interrupt latency, DPC latency or thread latency, respectively.

```
/*M*
//                  INTEL CORPORATION PROPRIETARY INFORMATION
//           This software is provided by Intel Corporation for your
//         internal use only for the purposes of performance measurement
//              and may not be otherwise used, copied or disclosed.
//       Copyright (c) 1998 Intel Corporation. All Rights Reserved.
//
//         The receipt of or possession of this program does not convey
//           any rights to reproduce its contents or to manufacture,
//           use, or sell anything that it may describe, in whole, or
//           in part, without specific consent of Intel Corporation.
//
//  Revision History: 06-15-98  ECR Initial version
//
//  Contents: SpinOnTimeStampCounter
//
//  Compiler Considerations: Only for Pentium(R), Pentium Pro, Pentium II,
//                           Celeron(TM) processors.
*M*/

/* . . . */

/*F*
//  Name:       SpinOnTimeStampCtr
//  Purpose:    Spin while reading the Pentium processor time stamp
//              counter until the specified time (cycle) is reached.
//  Context:    Called by driver to soak up time
//  Returns:    void
//  Parameters:
//      IN LARGE_INTEGER TimeToStop
*F*/
VOID
SpinOnTimeStampCtr (
   IN LARGE_INTEGER TimeToStop)
{
   LARGE_INTEGER CurrentTime;
   ULONG Lo;
   LONG Hi;

   do {
      _asm {
           push eax
           push edx
           _emit 0x0f
           _emit 0x31
           mov Lo, eax
           mov Hi, edx
           pop edx
           pop eax
           } /* _asm */
       CurrentTime.LowPart = Lo;
       CurrentTime.HighPart = Hi;
   } while (CurrentTime < TimeToStop);
} /* SpinOnTimeStampCtr */
```